Explain file generation can be partially automated by a

GENEXPLAIN <command>

command that creates a skeleton explain file and scans the source file *<command>.java* for additional fill-in-the-blank information. Here's the literal skeleton file *explain_skeleton.exp* generated by GENEXPLAIN with no <command> argument. Brace-delimited text reminds the author of the "proper" explain file form.

The {command name} {command type}

{command name} <{positional parameter 1}> ... [{keyword parameter 1}=] ...

{action verb} {high-level command description}

**Positional Parameters**

{parameter} {labelless parameter description} [default is {default value}]

{PARAMETER} {labelled parameter description} [default is {default value}]

**Keyword Parameters**

{keyword}= {parameter description} [default is {default value}]

**Command Switches**

In addition to system-defined command switches accepted by all NeXt-Midas commands, {command name} recognizes the following command-specific switches:

/{switch} {state switch description} [default is {default state}]

/{switch}= {value switch description} [default is {default value}]

**Messages**

When executed from a macro or as a background task, {command name} processes the following messages:

{msg ID} {message description}

{ID}=<{args}> {message description}

**Methods**

The following public methods are provided by the {command name} command object for direct access from the NeXt-Midas command line using the <object ID>.<method name>({parameter list}) syntax:

{method}({parameter list}) {method description}

**See Also**

{related commands}

When GENEXPLAIN is given a <command> argument, GENEXPLAIN checks the command dictionary to determine if <command> is already installed and, if it is, extracts the command entry. It also opens and reads the *<command>.java* source code. Based on contents of these non-mandatory inputs, GENEXPLAIN then replaces as much of the skeleton with command-specific information as possible. Here's how GENEXPLAIN works:

1. When <command> is supplied, the {command name} field in the skeleton is replaced by uppercased <command>.

2. When the command in installed already, the dictionary entry supplies the following additional information:

    a. Full command name, which is used in lieu of <command> if <command> is an abbreviation.

    b. Abbreviation point, which is revealed by showing mandatory characters of the full command name in uppercase and the remaining characters in lower case.

    c. Command type, which replaces {command type} field in skeleton.

    d. Positional parameter count, which is used to create the correct number of positional parameter description lines in the skeleton and placeholders in the command syntax synopsis. Ellipses ("...") identify commands that have a variable number of positional parameters.

    e. Distinction between labelled and unlabelled positional parameters. The {parameter} field for unlabelled positional parameters is replaced by <parameter {n}>, where {n} is the ordinal position. The {parameter} field for labelled parameters is replaced by the upppercased label name.

        NOTE...code and .cnf file in release 1.5.3 are inconsistent with regard to the possibility of and support for unlabelled positional parameters. If it is decided not to support them, explain file generation is simplied.

    f. Positional parameter default values, which are used to replace the {default value} fields in positional parameter description lines or to change the entire [default value is...] phrase with [no default]. ([no default] and [required parameter] conditions are not distinguished in the dictionary.)

GENEXPLAIN automatically writes two output files to the *../sys/exp* folder. These files are *<full command name>.exp*, which contains explain file text, and *<full command name>.java*, which defines the pro forma Java interface that links *<full command name>.exp*. Existing files are renamed *<full command name>.exp.save* and *<full command name>.java.save* first, if necessary.

The following example illustrates the explain file generation rules explained so far. Imagine the user gives the command

```
GENEXPLAIN MYCOMM
```

and the NeXt-Midas dictionary contains the entry

```
MYCOM*MAND      P,4+ ,IN=,,GREEN,TRIES=100,
```

Based on this information, GENEXPLAIN creates a file named *mycommand.exp* that starts with the following content:

```
The MYCOMmand primitive

    MYCOMmand <IN> <parameter 2> <parameter 3> <TRIES>

{action verb} {high-level command description}
```

**Positional Parameters**

| | |
|---|---|
| <IN> | {labelled parameter description} [no default] |
| <parameter 2> | {labelless parameter description} [no default] |
| <parameter 3> | {labelless parameter description} [default is GREEN] |
| <TRIES> | {labelled parameter description} [default is 100] |

The remainder of the embryonic explain file would contain only skeleton information. If, however, *mycommand.java* also existed, GENEXPLAIN would go on to read that file looking for the following "clues" for additional explain file information:

1. The first "sentence" in the "/** ... */" comment block that gives the Javadocs command summary replaces

   {action verb} {high-level command description}

2. Statements containing MA.get...() calls are analyzed to detect keyword parameters and switches. Keywords that match the labels on already determined positional parameters are ignored.

3. Whether or not a switch/keyword parameter has a default value is inferred from the MA.get...() calls. In cases where a literal default is supplied in the argument list, it is put into the explain file. Obviously, there are many ways that defaults might be assigned, and it's not the goal of GENEXPLAIN to be a comprehensive code analyzer. A future enhancement of this command could be to detect things like a default value supplied through an initialized variable rather than literal argument in a MA.get...() call.

4. Command-specific messages recognized by primitive commands are determined by analyzing processMessage(), if it exists, to find various string comparison operations performed on the name field of this method's Message argument. For example, Message.name.equals(...) and Message.name.startsWith(...) are commonly used in primitives to detect recognized messages.

Explain file generation for installed macro commands would need to be different. GENEXPLAIN would read the mycommand.mm file and check for common command constructs. For example, command-specific messages recognized by macro commands are determined by the processMessage procedure, if it exists, to find various string comparison operations performed on the macro's msg.name field. Most often, the macro contains commands like

```
(else)if msg.name eqs ...
```

to detect recognized messages.

To keep GENEXPLAIN from getting too complicated, the description of semantics of and data accompanying messages would be written manually.Similarly, the command methods truly intended for public access should be identified and documented by hand to avoid deluging the user with exhaustive information at the expense of clarity. Human intelligence is also required to make the associations appropriate to the See Also section.